搜索的应用

我们可以使用搜索来解决一些正解比较难解决的题目，可以取得**部分分数**，又是有运气好（当然不止运气，还有**好的优化也相当关键**）甚至可以得到全分。

在比赛是多拿几分就有可能从二等奖干倒一等奖！

所以让我们看看下面几个例子。

DFS 案例

CSP-J 2020 方格取数

分析

此题可能追求正解有一些难 (如果是dalao当我没说)。

数据范围很大 (1000×1000)，但也有很小的数据 (5×5)。

此时，~~对于我这种啥也不会蒟蒻~~我们可以使用**暴力搜索** (即**DFS**) 来骗拿到这道题的部分分数。

解法

我们设计一个叫做 `dfs(x, y, sum)` 的函数，来访问 (x, y)，到目前为止的路径和是 `sum`。此时分为两种情况：

- 如果找到终点 (n, m) 则比较路径和是否比答案更优，如果是，更新答案 `ans`。
- 否则，分别往上、下、右三个方向扩展。
    - 首先需要判断下一个位置没有超出范围，且没有被占用将其占用后，枚举下一个点（`sum` 要加上下个点的值）。

代码

```cpp
#include<bits/stdc++.h>
using namespace std;

int a[1010][1010], vis[1010][1010];

int n, m, ans = -11451419; // 初始值设置为很小的负数
void dfs(int x, int y, int sum){
    if(x == n && y == m){ // 当枚举到终点
        ans = max(sum, ans); // 如果 sum 大于 ans 则将 ans 更新为 sum
        return;
    }
    if(x > 1 && !vis[x - 1][y]){ // 枚举上方坐标
        vis[x - 1][y] = 1;
        dfs(x - 1, y, sum + a[x - 1][y]);
        vis[x - 1][y] = 0;
    }
    if(x <= n && !vis[x + 1][y]){ // 枚举下方坐标
        vis[x + 1][y] = 1;
        dfs(x + 1, y, sum + a[x + 1][y]);
        vis[x + 1][y] = 0;
    }
```

```
    if(y <= m && !vis[x][y + 1]){ // 枚举右边坐标
        vis[x][y + 1] = 1;
        dfs(x, y + 1, sum + a[x][y + 1]);
        vis[x][y + 1] = 0;
    }
}
int main() {
    cin >> n >> m;
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= m; j++)
            cin >> a[i][j];
    vis[1][1] = 1; // 将坐标 (1,1) 设为已经访问过了
    dfs(1, 1, a[1][1]); // 从 (1,1) 开始枚举，路径的和为 a[1][1]
    cout << ans;
}

```

此代码可以骗拿到 25 分。

BFS 案例

P1162 填涂颜色

题目就不用我copy下来了吧?

解法 && 解法

我们可以使用**类似洪水填充**的方法，观察下图（这不是文本么？），我们不难发现：

> 0 0 0 0 0 0
> 0 0 1 1 1 1
> 0 1 1 0 0 1
> 1 1 0 0 0 1
> 1 0 0 0 0 1
> 1 1 1 1 1 1

闭合区间内：被 `1` 围成一圈。
闭合区间外：**没有**被 `1` 围成一圈。

而且，此题的数据范围非常感人：

> 对于 100% 的数据，$1 \leq n \leq 30$。

所以，一个字：**暴搜**。

~~（这不是两个字吗？）~~

因此，我们可以定义一个 `bfs` 函数 `bfs(x,y)`，从 (x,y) 开始填充

- `bfs` 函数中，将 0 的格子传染成 3。
- 主函数中，对所有边缘上的格子都做一次 `bfs`。
- 最后输出时，3 输出成 0，1 不变，0 输出成 2。

代码

```cpp
#include <bits/stdc++.h>
using namespace std;
int n, a[35][35];
int fx[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
struct pos {
    int x, y;
    pos(int ax = 0, int ay = 0) {
        x = ax;
        y = ay;
    }
};
void bfs(int x, int y) { // 初始坐标是 (x,y)，也就是找到边缘的区域
    queue<pos> q;
    if(a[x][y] != 0) // 如果当前位置不是未填色的
        return;
    a[x][y] = 3; // 设置为边缘区域对应的位置
    q.push(pos(x, y));
    while(!q.empty()) {
        pos now = q.front();
        q.pop();
        for(int i = 0; i < 4; i++) {
            int xx = now.x + fx[i][0]; // 下一个点的坐标
            int yy = now.y + fx[i][1];
            if( xx < 1 || xx > n
             || yy < 1 || yy > n ) // 当下一个点的坐标超出范围
                continue;
            if(a[xx][yy] != 0) // 当下一个点不是 0
                continue;
            a[xx][yy] = 3; // 设置为边缘区域对应的位置
            q.push(pos(xx, yy));
        }
    }
}
int main() {
    cin >> n;
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= n; j++)
            cin >> a[i][j];

    for(int i = 1; i <= n; i++)bfs(1, i); // 遍历第 1 行的所有格子
    for(int i = 1; i <= n; i++)bfs(n, i); // 遍历最后一行的所有格子
    for(int i = 1; i <= n; i++)bfs(i, 1); // 遍历第 1 列的所有格子
    for(int i = 1; i <= n; i++)bfs(i, n); // 遍历最后一列的所有格子
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= n; j++) {
            if(a[i][j] == 3)
                cout << 0 << ' '; // 最后输出时的变换，详见课件
            else if(a[i][j] == 0)
                cout << 2 << ' ';
            else
                cout << 1 << ' ';
        }
        cout << endl;
```

```
    }
    return 0;
}

```